1

NASA Contractor Report 189598

ICASE Report No. 92-1

ICASE

PARALLELIZATION OF IRREGULARLY COUFLED REGULAR MESHES

Craig Chase
Kay Crowley
Joel Saltz
Anthony Reeves

Contract No. NAS1-18605
January 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Parallelization of Irregularly Coupled Regular Meshes[*]

Craig Chase[†]
Cornell University
School of Electrical Engineering
Ithaca, NY

Kay Crowley[‡]
Yale University
Department of Computer Science
New Haven, CT

Joel Saltz
ICASE
NASA Langley Research Center
Hampton, VA

Anthony Reeves
Cornell University
School of Electrical Engineering
Ithaca, NY

Abstract

Regular meshes are frequently used for modeling physical phenomena on both serial and parallel computers. One advantage of regular meshes is that efficient discretization schemes can be implemented in a straightforward manner. However, geometrically-complex objects, such as aircraft, cannot be easily described using a single regular mesh. Multiple interacting regular meshes are frequently used to describe complex geometries. Each mesh models a subregion of the physical domain. The meshes, or *subdomains*, can be processed in parallel, with periodic updates carried out to move information between the coupled meshes. In many cases, there are a relatively small number (one to a few dozen) subdomains, so that each subdomain may also be partitioned among several processors.

We outline a composite run-time/compile-time approach for supporting these problems efficiently on distributed-memory machines. This paper describes these methods in the context of a multiblock fluid dynamics problem developed at the NASA Langley Research Center.

[*] This work was supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605.

[†] Additional support for Chase has been provided by IBM Corporation

[‡] Additional support for Crowley has been provided by the NSF under NSF Grant ASC-8819374

1 Introduction

We are developing methods for porting programs with irregularly coupled regular meshes (ICRMs) commonly known as multiblock applications, to distributed-memory parallel computers. In order to ensure that our techniques are applicable to real-wo^rld problems, we have begun our research with a specific multiblock problem from the domain of computational fluid dynamics. Although our initial focus was multiblock CFD, we aim to produce methods that are applicable to all parallel codes that meet the following criteria:

- The data is divided into several interacting regions (typically called subdomains).
- There exists a computational phase in which work on each subdomain can be carried out independently.
- Data access patterns within each subdomain are regular.
- Communication between subdomains is limited to rectangular sections of data that are exchanged between subdomains.

In many problems there are at most a few dozen subdomains of varying sizes. We can assume that we will have to assign at least some of the subdomains to multiple processors, we must consequently be prepared to deal with multiple levels of parallelism in ICRM codes. A model of an ICRM application is shown in Figure 1. Typically ICRM applications have two levels of parallelism available. A coarse-grained parallelism is available for processing the subdomains concurrently. Each subdomain is a self-contained computation region that can, except for boundary conditions, be operated upon independently of the other subdomains. In addition, the computation for individual subdomains has fine-grain parallelism available. In order to achieve efficient execution of ICRM applications on distributed-memory multicomputers, both levels of parallelism must be exploited. Applying coarse-grained parallelism

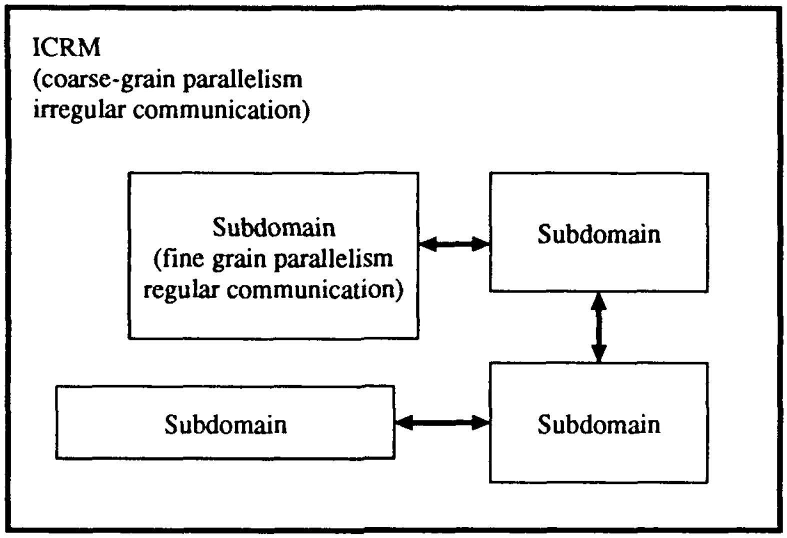

Figure 1: ICRM Application Model

will help to keep communication overhead to a manageable fraction of the computation time. However, since the number of subdomains is relatively small, particularly when compared to the number of processing elements in current distributed-memory multicomputers, the coarse-grained parallelism between subdomains will not provide sufficient parallel activity to keep all processors busy. The fine-grained parallelism within each subdomain must be used to fill this gap.

The methods we are developing to support ICRM applications are semi-automatic and include both compile-time and runtime support for partitioning and communication. We have developed and benchmarked on the Intel iPSC/860 the runtime support required to carry out the required patterns of interprocessor data motion. We have also developed a very rudimentary compiler prototype to embed this runtime support. The compiler produces, as output, Fortran 77 code that can be compiled and run on a distributed-memory parallel computer. This compiler prototype was built to experimentally define what will be needed to effectively support ICRM computations.

Our ultimate goal in this work is to provide language-level support for ICRMs in a general-purpose parallel language like Fortran D[FHK$^+$90]. We concentrate here on describing the functionality that must be added to such a language to handle ICRMs, and implementation techniques that efficiently support that functionality. In the course of our work, we have defined extensions to Fortran D that are useful for these problems; these are a means to an end, not the final product. Although we strongly believe that the functions provided by these new features will be critical for ICRM support, we believe that further work is needed to define appropriate syntax. We are currently collaborating with Rice to develop Fortran D extensions which capture the functionality we describe in this paper.

1.1 Problem Overview

The application we investigated is a problem from the domain of computational fluid dynamics. The serial code was developed by V. Vasta at the NASA Langley Research Center, and solves the thin-layer Navier-Stokes equations for a fluid flow over a three-dimensional surface with complex geometry. The problem geometry is decomposed into between one and a few dozen distinct regions, each of which is modeled with a regular, three-dimensional, rectangular grid. The boundary conditions of each region are enforced by simulating any of several situations including; viscous and inviscid walls, symmetry planes, extrapolation conditions, and interaction with an adjacent region. The size of each region (hereafter *subdomain*), its boundary conditions and adjacency information are loaded into the program at run time. For this application, the same program is run on all subdomains. However, different subroutines will be executed when applying the boundary conditions on different subdomains. In general, the code used to process each subdomain of an ICRM application may be different.

The sequence of activity for this program is as follows:

Read subdomain sizes, boundary conditions and simulation parameters,

Repeat (typically large number of times):

A. Apply boundary conditions to all subdomains,

B. Carry out computations on each subdomain.

The main body of the program consists of an outer sequential loop, and two inner parallel loops. Each of the inner loops iterate over the subdomains of the problem, the first applying boundary conditions (Step A), which may involve interaction with other subdomains, and the second loop advancing the physical simulation one time step in each subdomain (Step B). Partitioning of the parallel loops is the source of the coarse-grained parallelism for the application. Furthermore, within each iteration of the loop that implements Step B there is fine-grained parallelism available in the form of large parallel loops.

1.2 Compiler Overview

To investigate the extent to which ICRM applications can automatically be transformed for execution on a distributed-memory multicomputer, we designed a rudimentary compiler geared toward applying the specific set of transformations required by ICRMs. The compiler is built using the Sigma Toolkit [GLS$^+$91], which provides dependency and dataflow analysis. Sigma also provides a framework for applying transformations to programs and includes support for common dialects of Fortran, C and C++. As the main focus of the compiler was ICRM applications, a number of important compiler functions were not implemented. Rather than duplicate the efforts of ongoing or existing distributed-memory compiler projects, such as Fortran D [HKT91], Superb [ZBG86] [Ger89], and AL [Tse90], which have investigated many of the fundamental issues in distributed-memory compiling, our experimental compiler uses techniques which are complementary to these other ap-

Table 1: Compiler Transformations

	Parallelism	Communication
Between Subdomains	**Owner computes** loop bounds replaced with function calls	**Subarray Exchanges** replace copies of regular array sections with procedure calls
Within a Subdomain	**Owner computes** loop bounds replaced with function calls	**Overlap Cells** size of overlap determined at compile time procedure calls embedded to implement communication

proaches; applying specific transformations to those sections of the program that exhibit characteristic ICRM behavior.

The transformations performed by the compiler can be organized into four general categories, as shown in Table 1. The basic responsibilities of the compiler for ICRM applications are to handle the coarse grain parallelism between subdomains, the fine grain parallelism within a subdomain, and to ensure that the required communication takes place for both levels of parallelism. As our principal objectives were to determine the level of functionality required to handle ICRM applications, and to establish the potential for a compiler to automatically transform annotated ICRM programs for distributed-memory environments, the major focus of the compiler is to embed procedure calls to the enhanced PARTI runtime library.

Our transformations introduce both fine and coarse-grained parallelism into the program by enforcing the *owner computes* rule. Communication within a subdomain is implemented using overlap cells, utilizing both compile-time and runtime components. Communication between subdomains is provided through runtime support for exchanging regular array sections. Procedure calls to perform the data motion are inserted into the program by the compiler.

1.3 Organization

The remainder of this paper is organized as follows. In Section 2, the directives we use in the annotated version of the program are explained. Section 3 outlines the runtime library. Section 4 describes the parallelization of the computation for individual subdomains. In Section 5 we describe the techniques we developed for achieving parallel execution of multiple subdomains.

2 Fortran Directives

As part of our investigation into ICRM applications, we have identified the functionality needed to express data layout and organization on the processors. Integration of this functionality into the Fortran D language is currently underway. As a preliminary step, we have defined an experimental syntax for expressing this functionality in Fortran programs, and used this syntax to test our support for ICRMs. Although we feel that the expressive content of our directives is necessary for ICRM applications, the directives themselves are experimental, and unlikely to be adopted for implementation in Fortran D.

2.1 Subdomain Placement

The binding of subdomains to processors has important performance implications. Load balance plays a crucial role in determining computational efficiency. Since the amount of computation associated with each subdomain is directly proportional to the number of elements in the subdomain, good load balancing is achieved by binding processors to subdomains in a ratio proportional to their sizes. In our implementation, this mapping is under user control and is specified using program directives.

The principal abstraction for dealing with data placement is the *decomposition*. However, unlike Fortran D, where decompositions are bound to the entire processor set, we map decompositions to subsets of the processors. The mechanism for specifying this arrangement is a directive called *embed.* The *embed* directive binds a decomposition to a rectangular subregion of another decomposition. Any number of decompositions may be embedded into a single root decomposition. The root decomposition is mapped onto the entire set of physical processors. Embedded decompositions are mapped onto subsets of these processors based on the relative size, and location of the subregion in the root decomposition to which they are bound. This methodology can easily be extended recursively to support an arbitrary sequence of embeddings, although for most ICRM applications we are aware of, a two level decomposition hierarchy appears to be sufficient. The root level establishes a template onto which each subdomain can be mapped.

For the Navier-Stokes application, we use a one-dimensional decomposition for the root level, and embed 3-dimensional subdomains into it. For example, if two subdomains, one of size 10 × 10 × 10 and the other 5 × 5 × 10 were to be mapped onto the physical processing resource, a root-level decomposition of size 1,250 would be used. The first subdomain would be embedded into locations 1 through 1000 of this decomposition, and the second subdomain into locations 1001 through 1250. To clarify the distinction between the declaration of a decomposition and the specification of its size (which may be runtime dependent), we use two directives, *decomposition* and *shape*, to provide the same functionality as Fortran D's **decomposition**. This semantic partitioning allows us to conveniently declare an array of decompositions to hold the set of subdomains. The dimensionality and size of each of the decompositions in this array is determined dynamically by the *shape* directive. Although, in this example, the sizes are constants, in general, for an ICRM application, the subdomain sizes are not known until runtime.

2.2 Distributing Array Data

In our implementation, the arrays that make up each subdomain are distributed using the Fortran D *align* directive. However, since the number of subdomains, and their sizes, are not known until runtime, we allocate space using a single, one-dimensional work array. To make it possible to allocate space for multiple decompositions (or multiple elements of an array of decompositions) using a single work array, the *align* directive was extended to allow array reshaping. Our implementation of align supports the arbitrary reshaping of a region of memory into multidimensional, distributed arrays.

3 Run Time Support

The runtime support contains a number of PARTI procedures which carry out the bookkeeping needed to track the distributed arrays that describe ICRM problems. This runtime support is a generalized version of the runtime support described in [BSS91]. The major functions in the runtime library are listed in Table 2.

There are two principal data structures that are created and maintained in the runtime library. These data structures are *distributed array descriptors*, and *communication schedules*. The distributed array descriptor is a data structure that tracks a variety of attributes associated with each distributed array, including:

- array dimensionality and size,
- the number of overlap cells (see Section 4) in each dimension,
- array distribution in each dimension, and
- the set of processors to which the distributed array is mapped.

Communication schedules are data structures that describe how a specific data transfer is to be performed including:

Table 2: Runtime Library

Distribution Declarations	
create_decomposition	instantiates a decomposition
embed	maps decompositions to processors
distribute	establishes distribution pattern for decompositions
align	binds arrays to decompositions creates distributed array descriptor records overlap region sizes
Communication Primitives	
exchsched	makes schedule for overlap regions
subarraysched	makes schedule for subarray exchange
datamove	executes a schedule (communicates data)

- individual send and receive lists on each processor, and
- data access patterns for moving data between arrays and message buffers.

The communication schedule, or *schedule*, allows us to implement data motion as a two-phase process. Commonly known as Inspector/Executor, this methodology uses a preprocessing stage to determine the set of low-level communications primitives which must be used to transfer the data. A second stage then implements the data communication. This mechanism has been applied to irregular problems in the PARTI system [SCMB90], and to both regular and irregular problems as part of the *maparray* construct in Paragon [CCRS91].

Table 2 is organized into two components. The upper section of the table shows the primitives used to define the distribution of array data. The lower section lists the primitives used to perform data communication. These primitives can be used directly to program ICRM applications, or can be embedded into the program automatically.

The procedure **create_decomposition** defines a new decomposition with a given dimensionality and size in each dimension. The procedure **embed** implements the *embed* directive (see Section 2.1) and constrains the set of processors associated with a decomposition. A decomposition that has not been embedded into another decomposition is, by default, mapped to all processors.

The **distribute** procedure defines the type of distribution for each dimension of a decomposition (e.g. **BLOCK**, **CYCLIC** or **IRREGULAR**) and is used to implement the Fortran D *distribute* directive.

The **align** procedure implements the *align* directive and is used to associate arrays with decompositions and to create distributed array descriptors. The compiler determines the number of overlap cells for each array dimension and passes this information to **align**. **Align** writes the distributed array descriptor into a hash table, organized by array starting address. Using the hash table allows arrays to be passed as parameters between subroutines, transparently inheriting the distribution information from the calling procedure. Alternatively, the distribution data can, in some cases, be traced interprocedurally at compile time. Hiranandani *et al* define a process known as *reaching decompositions* which can be used to analyze array distribution both intra and inter procedurally [HKT91].

The communication primitives include a procedure **exchsched** which computes a schedule that is used to direct the filling of overlap cells along a given dimension of a distributed array. The schedule specifies required intra-processor data copying along with a set of send and receive calls.

The primitive **subarraysched** carries out the preprocessing required to copy the contents of a regular section, *source*, in one subdomain into a regular section, *destination*, in another subdomain. The interactions between subdomains for ICRM applications are limited to the exchange of regular subsections, as illustrated in Figure 2. The **subarraysched** primitive supports data moves between arbitrary rectangular sections of two subdomains, and can transpose the data along any dimension. **Subarraysched** can also copy the contents of a regular section in a given subdomain into another regular section

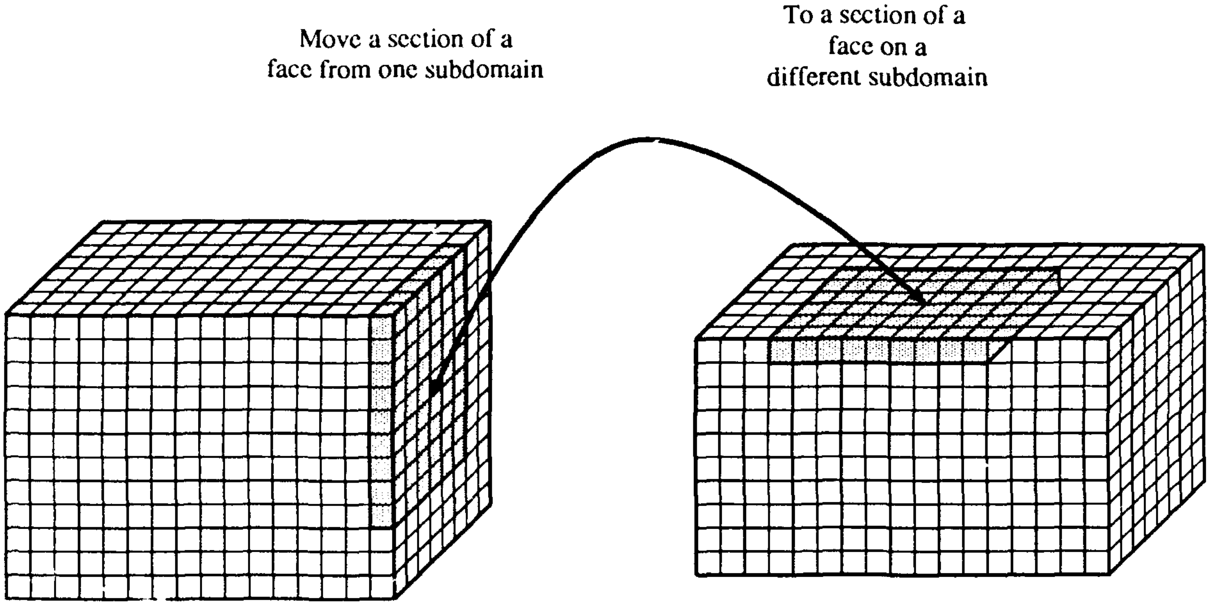
Move a section of a face from one subdomain
To a section of a face on a different subdomain

Figure 2: Moving Regular Subsections Between Subdomains

in the same subdomain. **Subarraysched** produces a schedule which specifies a pattern of intra-processor data transfers along with a set of send and receive calls. The primitive **subarraysched** executes on each processor. On a given processor P, **subarraysched** must find out whether it owns any portion of *source*. If P does own some portion, *source*$_P$, of *source*, **subarraysched** must calculate the processors to which various subsets of *source*$_P$ will have to be sent. **Subarraysched** must also calculate whether processor P owns any portion of *destination* and, if so, it must prepare to receive the appropriate messages.

Schedules produced by **exchsched** and **subarraysched** are employed by a primitive called **datamove** that carries out communication and intra-processor data copying.

4 Computation Within a Subdomain

The computation within a subdomain requires predominantly near-neighbor communi-

```
do k=1,kmx
do j=1,jmx
do i=1,imx
    qs = (skx(i,j,k) + sky(i,j,k) +
         skz(i,j,k))/ra
    ra = 0.5*(w(i,j,k+1) + w(i,j,k))
    hs(i,j,k) = qs*ra
    ra = 0.5*(w(i,j+1,k) + w(i,j+1,k))
    gs(i,j,k) = qs*ra
    ra = 0.5*(w(i+1,j,k) + w(i,j,k))
    fs(i,j,k) = qs*ra
enddo
enddo
enddo
```

Figure 3: Example Code for Sweep Over a Single Subdomain

cation. A typical loop nest for this component of an ICRM application is shown in Figure 3. The loop nest is computationally intensive, with no loop-carried data dependencies. Because the communication is regular, this code can be efficiently handled by the overlap cell method described by Gerndt in [Ger90]. Our compiler transforms this code as follows:

- Overlap cells are determined by scanning every subroutine in the procedure and accumulating the data in an interprocedural analysis phase of the compiler. When two subroutines have different overlap cell requirements, the maximum of the two values is used. The final value for the number of overlap cells for each dimension of every array must be a constant.

- Local array sizes are determined dynamically at subroutine boundaries. The array sizes are computed by a function in the run time library, and includes the extra memory required for the overlap cells.

- Loops are partitioned to enforce the owner computes rule within the loop body. For this transformation, the compiler identifies an array appearing on the left hand side of an assignment statement for which the loop-index is used as a subscript. This loop is then partitioned in the same manner as the array. For multidimensional arrays, the compiler identifies a specific dimension of an array for each loop in the loop nest.

4.1 Performance

Figure 4 shows the performance obtained while processing a single 64,000 element subdomain. Figure 5 shows the same data normalized by the number of processors. The data was collected using an iPSC/860 multicomputer processing a single $40 \times 40 \times 40$ subdomain. The timings were made from a single routine which is representative of the computation behavior of the program while processing an individual subdomain. The curve labeled "Actual" shows the performance, in megaflops, obtained for a single invocation of the subroutine. The "Optimistic" curve shows the performance that results when the time spent computing the communication schedules is excluded. Since schedules can be reused, this cost can be amortized over several invocations of the subroutine. The optimistic curve reflects the asymptotic performance for several iterations of this routine.

The "Ideal" curve includes only the message-passing time, and excludes the time required to create communication schedules, and the time spent reorganizing the data. For a multi-dimensional array, the elements that must be transferred to fill the overlap cells will not, in general, be in a contiguous section of memory. To transfer this data between processors on the iPSC/860, it must be first copied into a local buffer. After transmission, the data is again reorganized as it is placed into the overlap cells. The "Ideal" curve excludes the time spent packing and unpacking data. Since communication is always required for a distributed-memory implementation, this curve demonstrates the maximum possible performance for this loop given the bandwidth and communication latencies of the iPSC/860.

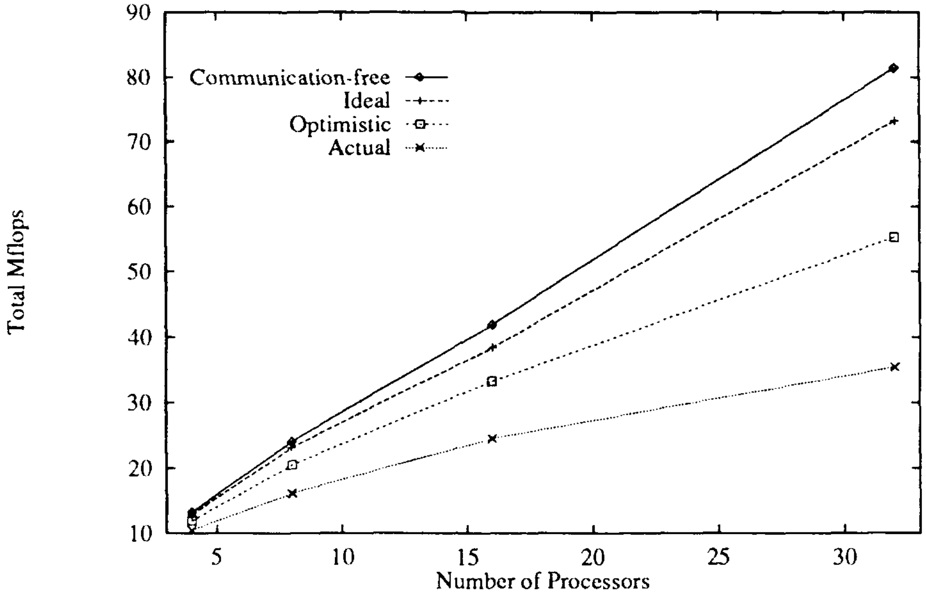

Figure 4: iPSC/860 Single-Subdomain Performance

The curve labeled "Communication-free" shows the computation rate obtained when no communication takes place. As the partition size on each processor decreases, the computation rate on each node also decreases. This effect is attributable to the increased relative cost of loop overhead and pipeline setup time. This curve demonstrates that even when communication effects are excluded, a large grain size will result in better overall performance. This data indicates the upper bound on the performance imposed by the application program code and the if77 compiler.

5 Supporting Multiple Subdomains

An important characteristic of ICRM applications is the relative independence of the subdomains. Much of the computation for a subdomain can be performed in parallel with the processing of other subdomains. As Figure 5 illustrates, there is a potential for much

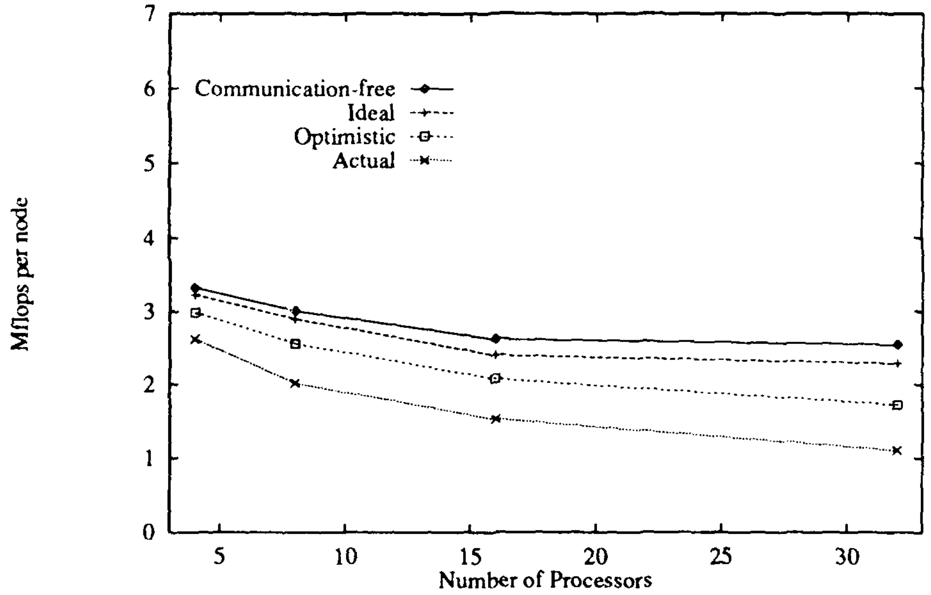

Figure 5: Per Node Single-Subdomain Performance

higher overall performance by partitioning the set of processors, and binding a relatively small number of processors to each subdomain. The cost of this approach is periodic synchronization between those subdomains which must exchange data.

5.1 Inter-Subdomain Communication

Although the processing of individual subdomains exhibits regular communication, interaction between subdomains is irregular. An illustration of the sort of communication that is required is shown in Figure 6. The figure shows two subdomains, one which models the airflow around a wing and another which models the region around a control surface on the wing. In this problem, there are two boundary conditions which require inter-subdomain communication. These boundary conditions consist of segments along exterior edges of the grids that, in the problem geometry, are adjacent. Although the sections are rectangular, the beginning and ending points of the sections are not determined until runtime. In general, the adjacency information for an ICRM is problem specific, and not determined until runtime. However, an efficient implementation should be able to take advantage of the fact that communication is limited to the exchange of rectangular sections of data.

Transforming an ICRM application to efficiently handle this type of data communication begins by identifying those locations in the program which require data transfer between subdomains. Our implementation recognizes code that performs regular data moves between arrays by simple symbolic analysis of array subscripts and checking the dataflow pattern in loop bodies. When the compiler detects that a regular section of an array is being transferred into another array, it removes the assignment statements from the loop body and inserts procedure calls to implement the data motion. Since the runtime library can move regular sections of data between subdomains, or within the same subdomain, this technique is safe for any parallel loop (*i.e.* a loop with no loop-carried dependencies).

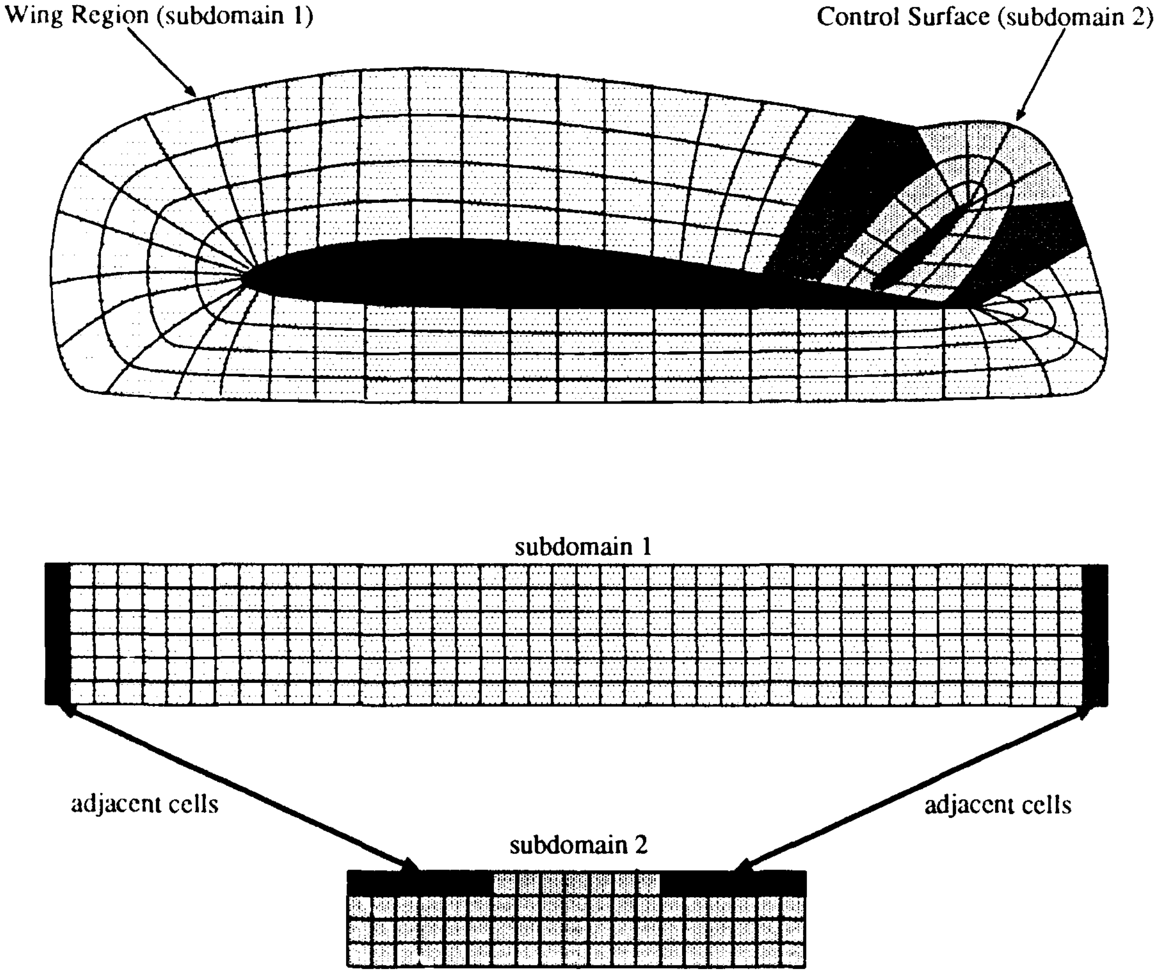

Figure 6: Data Movement Between Subdomains

Table 3: Inter-Subdomain Communication

Processors	Transfer	Scheduling	Total
4	11ms	32ms	43ms
8	6.3ms	32ms	39ms
16	4.5ms	28ms	33ms
32	2.5ms	25ms	28ms

5.2 Performance

We applied our methodology for inter-subdomain data transfers to a simplified version of the Navier-Stokes application and benchmarked the resulting code on the iPSC/860. Table 3 shows the time required to communicate data between two $40 \times 40 \times 40$ subdomains for different processor sizes. Each subdomain is placed on half of the processors, and one 40×40 face is transferred from subdomain 1 to subdomain 2. The column labeled "Transfer" gives the time required to actually transfer the data. The "Scheduling" column shows the time required to compute the communication schedules. Since communication schedules can be reused several times, the asymptotic performance is the data transfer rate.

As a point of reference, the time required to complete one iteration of the subroutine benchmarked in Section 4.1 was on the order of a few hundred milliseconds; ranging from 230ms on 32 processors to 760ms for 4 processors. The inter-subdomain communication time is an order of magnitude less than the computation time for each iteration and is not an impediment to processing multiple subdomains in parallel.

5.3 Partitioning for Coarse-Grain Parallelism

The parallelism between subdomains can be obtained by introducing partitioning at several levels in the program. Our implementation is based on introducing partitioning at a relatively low level. With the exception of standard library routines, such as **sqrt**, subroutine calls are run on every processor. Within a subroutine, loops are partitioned to mirror the array distribution, as described in Section 4. This loop partitioning results in parallel execution of the loop on all processors bound to the subdomain. In addition, concurrent processing of multiple subdomains also develops, as processors bypass the loops which iterate over non-local subdomains. This method is quite effective at extracting the coarse-grained parallelism available in the Navier-Stokes application because of the large amount of computation within each subdomain. Since a typical loop over an individual subdomain requires several tens or perhaps hundreds of milliseconds on the iPSC/860, the overhead associated with subroutine calls and runtime tests to check locality is insignificant.

Our straightforward approach is safe, and will result in correct execution on distributed-memory multicomputers, but it is not especially aggressive. More sophisticated techniques may be required to extract the coarse-grained parallelism in programs with many, relatively small subdomains. The loop in Figure 7 shows a simplified version of one of the main loops for the Navier-Stokes application. This loop iterates over a set of meshes (subdomains) in sequence. For each mesh, three subroutines are called. The parameters to the subroutines are a section of the array X, and the sizes for each dimension of the current mesh. As described in Section 2.2, a single, one-dimensional space array is used to hold the data for all subdomains. In our implementation, every processor executes this loop serially, and executes every subroutine call. For any given subdomain, m, some processors will participate in the computation of the subroutines, and others will simply fall through the loops; performing no iterations because they store none of the elements of m.

More efficient parallelization is possible if the unnecessary subroutine calls can be avoided on individual processors. A compile-time methodology for this can be based on interprocedural regular section analysis. By performing regular section analysis, such as that described in [HK90], [HK91] it may be possible to determine that a regular region of the array X is accessed within the individual subroutines. Further symbolic analysis can

```
do m = 1, num_domains
    call navier(X(mesh(m)), isize(m), jsize(m), ksize(m))
    call flux(X(mesh(m)), isize(m), jsize(m), ksize(m))
    call solve(X(mesh(m)), isize(m), jsize(m), ksize(m))
enddo
. . .
subroutine navier(X, isz, jsz, ksz)
dimension X(isz,jsz,ksz)
. . .
end
```

Figure 7: Loop Over Subdomains

then be applied to associate this section of X with the subdomain to which it has been aligned. Note that this test may require interprocedural analysis since the distribution of X may have occurred in a different subroutine from the loop shown in Figure 7. Once the compiler has determined that each subroutine invocation accesses only a single subdomain, it may partition the loop according to which subdomains are local.

An alternative approach to partitioning this loop could be based on runtime preprocessing. Since the loop shown in Figure 7 is executed many times as part of an outer sequential loop (see Section 1.1), the cost of determining the loop partitioning at runtime will likely be an insignificant part of the total execution time. However, even with a runtime approach, some analysis must be performed to ensure that the partitioning remains valid as this loop is reexecuted at each iteration of the outer loop.

Both of these approaches require interprocedural and symbolic analysis that may extend the limit of what seems reasonable to expect from the current generation of compilers. User input, in the form of additional directives, may be required in order to partition loops such as the one shown in Figure 7. Furthermore, since a high-level of coarse-grained parallelism can still be obtained on large, numerically-intensive programs even when this loop is executed sequentially on all processors, the most effective method for extracting the inter-subdomain parallelism from ICRM applications is an open question.

6 Conclusions

We have developed methods for efficiently executing ICRM applications on distributed memory multicomputers. These applications are an important class of scientific programs with computational behavior requiring specialized support not available in the current set of distributed-memory compilers. The fundamental aspects of this class of applications that we have addressed include:

- Identifying the set of functionality to be introduced at the language level for programming ICRMs.
- Developing a methodology for maintaining several interacting subdomains, each distributed on a subset of the total processors.
- Providing communication support both within a subdomain and between subdomains.
- Identifying the compile-time requirements for embedding the communication support, and for extracting parallelism (both fine-grained and coarse-grained) from ICRM applications.

The efficacy of our approach has been verified using a rudimentary compiler which implements a set of highly specialized program transformations, and embeds procedure calls to implement data motion. A runtime library has been developed for the iPSC/860 multicomputer. This library implements the core set of functionality required by ICRMs, and has been tested using an applications program developed at the NASA Langley Research Center. The communications overhead imposed by the runtime support has been shown to not be prohibitive to achieving good performance on the iPSC/860.

Acknowledgements

The authors would like to thank Chuck Koelbel, Lorie Liebrock, Reinhard von Hanxleden and Ken Kennedy for extremely useful discussions about compiler support of ICRMs and how this functionality could be integrated into Fortran D. We would like to thank Chuck Koelbel for insight into the (eventual) use of interprocedural analysis in the parallelization of ICRM computations.

We would also like to thank Dennis Gannon for providing us with an early release of the Sigma toolkit.

References

[BSS91] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory architectures. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[CCRS91] Craig Chase, Alex Cheung, Anthony Reeves, and Mark Smith. Programming for scientific computation using communication structures. In *International Conference on Parallel Processing*, 1991.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[Ger89] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Institute for Statistics and Computer Science, University of Vienna, 1989.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice & Experience*, 1990.

[GLS+91] Dennis Gannon, Jenq Kuen Lee, Bruce Shei, Sekhar Sarukkai, Srinivas Narayana, Neelakantan Sundaresan, Daya Attapatu, and Francois Bodin. SIGMA II: A tool kit for building parallelizing compilers and performance analysis systems. Technical report, Indiana University, 1991.

[HK90] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for fortran D on mimd distributed-memory machines. In *Supercomputing '91, Albuquerue, New Mexico*, November 1991.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[Tse90] P. S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *SIGPLAN '90*, White Plains, NY, June 1990.

[ZBG86] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
January 1992

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
PARALLELIZATION OF IRREGULARLY COUPLED REGULAR MESHES

5. FUNDING NUMBERS
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
Craig Chase, Kay Crowley, Joel Saltz and Anthony Reeves

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Repcrt No. 92-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-189598
ICASE Report No. 92-1

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
International Supercomputing Conference

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61,64

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Regular meshes are frequentlv used for modeling physical phenomena on both serial and parallel computers. One advantage of regular meshes is that efficient discretization schemes can be implemented in a straightforward manner. However, geometrically-complex objects, such as aircraft, cannot be easily described using a single regular mesh. Multiple interacting regular meshes are frequently used to describe complex geometries. Each mesh models a subregion of the physical domain. The meshes, or **subdomains**, can be processed in parallel, with periodic updates carried out to move information between the coupled meshes. In many cases, there are a relatively small number (one to a few dozen) subdomains, so that each subdomain may also be partitioned among several processors.

We outline a composite run-time/compile-time approach for supporting these problems efficiently on distributed-memory machines. This paper describes these methods in the context of a multiblock fluid dynamics problem developed at the NASA Langley Research Center.

14. SUBJECT TERMS
Distributed memory compiler, Navier Stokes, Block structured, Fortran D

15. NUMBER OF PAGES
25

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

NASA Langley, 1992